Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter, ianni, roman, tompits}@kr.tuwien.ac.at

Abstract. Towards providing a suitable tool for building the Rule Layer of the Semantic Web, HEX-programs have been introduced as a special kind of logic programs featuring capabilities for higher-order reasoning, interfacing with external sources of computation, and default negation. Their semantics is based on the notion of answer sets, providing a transparent interoperability with the Ontology Layer of the Semantic Web and full declarativity. In this paper, we identify classes of HEX-programs feasible for implementation yet keeping the desirable advantages of the full language. A general method for combining and evaluating sub-programs belonging to arbitrary classes is introduced, thus enlarging the variety of programs whose execution is practicable. Implementation activity on the current prototype is also reported.

1 Introduction

For the realization of the Semantic Web, the integration of different layers of its conceived architecture is a fundamental issue. In particular, the integration of *rules* and *ontologies* is currently under investigation, and many proposals in this direction have been made. They range from homogeneous approaches, in which rules and ontologies are combined in the same logical language (e.g., in SWRL and DLP [16, 13]), to hybrid approaches in which the predicates of the rules and the ontology are distinguished and suitable interfacing between them is facilitated, like, e.g., [10, 8, 25, 15] (see also [1] for a survey). While the former approaches provide a seamless semantic integration of rules and ontologies, they suffer from problems concerning either limited expressiveness or undecidability, because of the interaction between rules and ontologies. Furthermore, they are not (or only to a limited extent) capable of dealing with ontologies having different formats and semantics (e.g., RDF and OWL) at the same time. This can be handled, in a fully transparent way, by the approaches which keep rules and ontologies separate. Ontologies are treated as external sources of information, which are accessed by rules that also may provide input to the ontologies. In view of the well-defined interfaces, the precise semantic definition of ontologies and their actual structure does not need to be known. This in particular facilitates ontology access as a Web service, where also privacy issues might be involved (e.g., a customer taxonomy in the financial domain).

In previous work [8], HEX-programs were introduced as a generic rule-based language fostering a hybrid integration approach towards implementing the Rule Layer of the Semantic Web ("HEX" stands for *higher-order with external atoms*). They are based on

Y. Sure and J. Domingue (Eds.): ESWC 2006, LNCS 4011, pp. 273–287, 2006.

nonmonotonic logic programs, which support constructs such as default negation, under the answer-set semantics, which underlies the generic *answer-set programming* (ASP) paradigm for declarative problem solving. The latter has proven useful in a variety of domains, including planning, diagnosis, information integration, and reasoning about inheritance, and is based on the idea that problems are encoded in terms of programs such that the solutions of the former are given by the models (the "answer sets") of the latter. The availability of default negation allows an adequate handling of conflict resolution, non-determinism, and dealing with incomplete information, among other things.

HEX-programs compensate limitations of ASP by permitting *external atoms* as well as *higher-order atoms*. They emerged as a generalization of *dl-programs* [10], which themselves have been introduced as an extension of standard ASP, by allowing a coupling with description-logic knowledge bases, in the form of *dl-atoms*. In HEX-programs, however, an interfacing with *arbitrary* external computations is realized. That is to say, the truth of an external atom is determined by an external source of computation. For example, the rule $triple(X, Y, Z) \leftarrow \&rdf[url](X, Y, Z)$ imports external RDF theories taking values from the external predicate $\&rdf$. The latter extracts RDF statements from a given set of URLs (encoded in the predicate *url*) in form of a set of "reified" ternary assertions. As another example, $C(X) \leftarrow triple(X, Y, Z), (X, rdf{:}type, C), not\ filter(C)$ converts triples to facts of a respective type, unless this type is filtered. Here, $C(X)$ is a higher-order atom, where C ranges over predicates constrained by $not\ filter(C)$.

HEX-programs are attractive since they have a fully declarative semantics, and allow for convenient knowledge representation in a modular fashion without bothering about the order of rules or literals in the bodies of rules of a program. However, the presence of external and higher-order atoms raises some technical difficulties for building implemented systems, given the following design goals which should be kept:

Full declarativity. This would mean that the user must be enabled to exploit external calls ignoring the exact moment an evaluation algorithm will invoke an external reasoner. So external calls must be, although parametric, stateless.

Potentially infinite universe of individuals. Current ASP solvers work under the assumption of a given, finite universe of constants. This ensures termination of evaluation algorithms (which are based on grounding), but is a non-practical setting if actual external knowledge must be brought inside the rule layer. Therefore, suitable methods must be devised for bringing finite amounts of new symbols into play while keeping decidability of the formalism.

Expressive external atoms. Interfacing external sources should support (at least) the exchange of predicates, and not only of constants (i.e., individuals). However, the generic notion of an external atom permits that its evaluation depends on the interpretation as a whole. For a practical realization, this quickly gets infeasible. Therefore, restricted yet still expressive classes of external atoms need to be identified.

These problems are nontrivial and require careful attention. Our main contributions are briefly summarized as follows.

We consider meaningful classes of HEX-programs, which emerge from reasonable (syntactic and semantic) conditions, leading to a categorization of HEX-programs. They include a notion of stratification, laid out in Section 3.1, which is more liberal than previous proposals for fragments of the language (e.g., as for HiLog programs [21]), as

well as syntactic restrictions in terms of safety conditions for the rules, as discussed in Section 3.2. Furthermore, we consider restricted external predicates with additional semantic annotation which includes types of arguments and properties such as monotonicity, anti-monotonicity, or linearity.

Section 3.3 introduces a method of decomposing HEX-programs into separate modules with distinct features regarding their evaluation algorithm, and Section 3.4 discusses strategies for computing the models of HEX-programs by hierarchically evaluating their decomposed modules.

Finally, we have implemented a prototype of HEX-programs. The current implementation features dl-atoms and RDF-atoms for accessing OWL and RDF ontologies, respectively, but also provides a tool kit for programming customized external predicates. The prototype actually subsumes a prototype for dl-programs [10] we built earlier.

Our results are important towards the effective realization of a fully declarative language which integrates rules and ontologies. While targeted for HEX-programs, our methods and techniques may be applied to other, similar languages and frameworks as well. Indeed, HEX-programs model various formalisms in different domains [8], and special external atoms (inspired by [10]) are important features of other recent declarative rule formalisms for the Semantic Web [25, 15, 24].

2 HEX-Programs

In this section, we briefly recall HEX-programs; for further background, see [8].

Before describing syntax and semantics, we consider an example to give the flavor of the formalism. An interesting application scenario where several features of HEX-programs come into play is *ontology alignment*. Merging knowledge from different sources in the context of the Semantic Web is a crucial task. To avoid inconsistencies which arise in merging, it is important to diagnose the source of such inconsistencies and to propose a "repaired" version of the merged ontology. In general, given an entailment operator $\models$ and two theories T_1 and T_2, we want to find some theory $rep(T_1 \cup T_2)$ which, if possible, is consistent (relative to $\models$). Usually, *rep* is defined according to some customized criterion, so that to save as much knowledge as possible from T_1 and T_2. Also, *rep* can be nondeterministic and admitting more than one possible solution.

HEX-programs allow to define the relation $\models$ according to a range of possibilities; as well, HEX-programs are a useful tool for modeling and customizing the *rep* operator. How HEX coding can achieve these goals is sketched in the following program, P_{ex}:

$$triple(X, Y, Z) \leftarrow url(U), \&rdf[U](X, Y, Z); \tag{1}$$

$$proposition(P) \leftarrow triple(P, rdf{:}type, rdf{:}Statement); \tag{2}$$

$$pick(P) \vee drop(P) \leftarrow proposition(P); \tag{3}$$

$$pick(P) \leftarrow axiomatic(P); \tag{4}$$

$$C(rdf{:}type, X) \leftarrow picked(X, rdf{:}type, C); \tag{5}$$

$$D(rdf{:}type, X) \leftarrow picked(C, rdf{:}subClassOf, D), C(rdf{:}type, X); \tag{6}$$

$$picked(X,Y,Z) \leftarrow pick(P), triple(P, rdf{:}subject, X), \tag{7}$$
$$triple(P, rdf{:}predicate, Y), \tag{8}$$
$$triple(P, rdf{:}object, Z),\ not\ filter(P); \tag{9}$$
$$\leftarrow \&inconsistent[picked]. \tag{10}$$

P_{ex} illustrates some features of HEX programs, such as:

Importing external theories. Rule (1) makes use of an external predicate *&RDF* intended to extract knowledge from a given set of URLs.

Searching in the space of assertions. Rules (2) and (4) choose nondeterministically which propositions have to be included in the merged theory and which not. These rules take advantage of disjunction in order to generate a space of choices.

Translating and manipulating reified assertions. E.g., it is possible to choose how to put RDF triples (possibly including OWL assertions) in an easier manipulatable and readable format, making selected propositions true as with rules (5) and (7).

Defining ontology semantics. The operator $\models$ can be defined in terms of rules and constraints expressed in the language itself, as with rule (6) or constraint (10). The external predicate *&inconsistent* takes for input a set of assertions and establishes through an external reasoner whether the underlying theory is inconsistent.

HEX-programs are built on mutually disjoint sets $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ of *constant names*, *variable names*, and *external predicate names*, respectively. Unless stated otherwise, elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are denoted with first letter in upper case (resp., lower case); elements from $\mathcal{G}$ are prefixed with "&".[1] Constant names serve both as individual and predicate names. Importantly, $\mathcal{C}$ may be infinite.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. A *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \ldots, Y_n)$, where $Y_0, \ldots, Y_n$ are terms; $n \geq 0$ is its *arity*. Intuitively, Y_0 is the predicate name; we thus also use the familiar notation $Y_0(Y_1, \ldots, Y_n)$. The atom is *ordinary*, if Y_0 is a constant. For example, $(x, rdf{:}type, c)$ and $node(X)$ are ordinary atoms, while $D(a, b)$ is a higher-order atom. An *external atom* is of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m), \tag{11}$$

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called *input list* and *output list*, respectively), and $\&g \in \mathcal{G}$ is an external predicate name. We assume that $\&g$ has fixed lengths $in(\&g) = n$ and $out(\&g) = m$, respectively. Intuitively, an external atom provides a way for deciding the truth value of an output tuple depending on the extension of a set of input predicates.

Example 1. The external atom $\&reach[edge, a](X)$ may compute the nodes reachable in the graph *edge* from the node a. Here, $in(\&reach) = 2$ and $out(\&reach) = 1$. □

A HEX-*program*, P, is a finite set of rules of form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, not\,\beta_{n+1}, \ldots, not\,\beta_m, \tag{12}$$

[1] In [8], "#" is used instead of "&"; we make the change to be in accord with the syntax of the prototype system.

where $m, k \geq 0$, $\alpha_1, \ldots, \alpha_k$ are atoms, and $\beta_1, \ldots, \beta_m$ are either atoms or external atoms. For a rule r as in (12), we define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then r is a *constraint*, and if $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a *fact*; r is *ordinary*, if it contains only ordinary atoms, and P is ordinary, if all rules in it are ordinary.

The semantics of HEX-programs generalizes the answer-set semantics [12], and is defined using the *FLP-reduct* [11], which is more elegant than the traditional reduct and ensures minimality of answer sets.

The *Herbrand base* of a HEX-program P, denoted HB_P, is the set of all possible ground versions of atoms and external atoms occurring in P obtained by replacing variables with constants from C. The grounding of a rule r, $grnd(r)$, is defined accordingly, and the grounding of program P is $grnd(P) = \bigcup_{r \in P} grnd(r)$.

For example, for $C = \{edge, arc, a, b\}$, ground instances of $E(X, b)$ are, for instance, $edge(a, b)$, $arc(a, b)$, and $arc(arc, b)$; ground instances of $\&reach[edge, N](X)$ are $\&reach[edge, edge](a)$, $\&reach[edge, arc](b)$, and $\&reach[edge, edge](edge)$, etc.

An *interpretation relative to P* is any subset $I \subseteq HB_P$ containing only atoms. We say that I is a *model* of atom $a \in HB_P$, denoted $I \models a$, if $a \in I$. With every external predicate name $\&g \in \mathcal{G}$ we associate an $(n+m+1)$-ary Boolean function $f_{\&g}$ (called *oracle function*) assigning each tuple $(I, y_1 \ldots, y_n, x_1, \ldots, x_m)$ either 0 or 1, where $n = in(\&g)$, $m = out(\&g)$, $I \subseteq HB_P$, and $x_i, y_j \in C$. We say that $I \subseteq HB_P$ is a *model* of a ground external atom $a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m)$, denoted $I \models a$, iff $f_{\&g}(I, y_1 \ldots, y_n, x_1, \ldots, x_m) = 1$.

Example 2. Associate with $\&reach$ a function $f_{\&reach}$ such that $f_{\&reach}(I, E, A, B) = 1$ iff B is reachable in the graph E from A. Let $I = \{e(b, c), e(c, d)\}$. Then, I is a model of $\&reach[e, b](d)$ since $f_{\&reach}(I, e, b, d) = 1$. □

Let r be a ground rule. We define (i) $I \models H(r)$ iff there is some $a \in H(r)$ such that $I \models a$, (ii) $I \models B(r)$ iff $I \models a$ for all $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$, and (iii) $I \models r$ iff $I \models H(r)$ whenever $I \models B(r)$. We say that I is a *model* of a HEX-program P, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$.

The *FLP-reduct* [11] of P with respect to $I \subseteq HB_P$, denoted fP^I, is the set of all $r \in grnd(P)$ such that $I \models B(r)$. $I \subseteq HB_P$ is an *answer set of P* iff I is a minimal model of fP^I. By $AS(P)$ we denote the set of all answer sets of P.

Example 3. Consider the program P_{ex} from the above, together with the set F of facts $\{url(\text{“}\texttt{http://www.polleres.net/foaf.rdf}\text{”}), url(\text{“}\texttt{http://www.gibbi.com/foaf.rdf}\text{”})\}$.

Suppose that the two URLs contain the triples $(gibbi, hasHomepage, url)$ and $(gibbi, hasHomepage, url2)$, respectively, and that $\&inconsistent$ is coupled with an external reasoner such that the property *hasHomepage* is enforced to be single valued. Then, $P_{ex} \cup F$ has two answer sets, one containing the fact $picked(gibbi, hasHomepage, url)$ and the other the fact $picked(gibbi, hasHomepage, url2)$. Note that the policy for picking and/or dropping propositions can be customized by changing the HEX-program at hand. □

3 Decomposition of HEX-Programs

Although the semantics of HEX-programs is well-defined, some practical issues remain and need further attention:

1. It is impractical to define the semantics of each external predicate by means of a Boolean function. Also, most of the external predicates encountered do not depend on the value of the whole interpretation but only on the extensions of predicates specified in the input. We thus introduce a model in which external predicates are associated with functions whose input arguments are typed.
2. Many external predicates have regular behavior. For instance, their evaluation function may be monotonic with respect to the given input, like for most of the dl-atoms introduced in [10]. These kinds of behaviors need to be formalized so that they can be exploited for efficient evaluation of HEX-programs.
3. It is important to find a notion of mutual predicate dependency and stratification that accommodates the new sorts of introduced constructs, in order to tailor efficient evaluation algorithms.
4. Although the semantics of HEX-programs fosters a possibly infinite Herbrand universe, it is important to bound the number of symbols that have to be actually taken into account, by means of adequate restrictions. In any case, the assumption that oracle functions are decidable is kept, but they may have an infinite input domain and co-domain.

To take the first two points into account, we introduce the following concept:

Definition 1. *Let &g be an external predicate, $f_{\&g}$ its oracle function, I an interpretation, and $q \in C$. Furthermore, we assume that:*

- *in(&g) = n and out(&g) = m;*
- *&g is associated with a* type signature $(t_1, \ldots, t_n)$*, where each t_i is the* type *associated with position i in the input list of &g. A type is either* c *or a nonnegative integer value. If t_i is* c*, then we assume the i-th input of $F_{\&g}$ is a constant, otherwise we assume that the i-th input of $F_{\&g}$ ranges over relations of arity t_i.*
- *For $\mathsf{a} \geq 0$, D_{a} is the family of all sets of atoms of arity* a *and $D_{\mathsf{c}} = C$; and*
- *$\Pi_{\mathsf{a}}(I, q)$ is the set of all atoms belonging to I having q as predicate name and arity* a*, whereas $\Pi_{\mathsf{c}}(I, q) = q$.*

Then, $F_{\&g} : D_{t_1} \times \cdots \times D_{t_n} \rightarrow D_{m-1}$ is an extensional evaluation function *iff* $(a_1, \ldots, a_m) \in F_{\&g}(\Pi_{t_1}(I, p_1), \ldots, \Pi_{t_n}(I, p_n))$ *precisely if* $f_{\&g}(I, p_1, \ldots, p_n, a_1, \ldots, a_m) = 1$.

We will call the external predicates associated with an extensional evaluation function and a type signature *typed*. Unless specified otherwise, we will assume in what follows to deal with typed external predicates only.

An evaluation function is a means for introducing an explicit relationship between input and output values of an external atom, and for expressing restrictions on the type of input values. Actual parameters inside external atoms express how, in the context of a given rule, input arguments are given in order to compute the output relation.

Example 4. Associate with predicate &*reach* an evaluation function $F_{\&reach}$ and a type signature $(2, 0)$ such that $F_{\&reach}(\Pi_2(I, E), A) = B$, where B is the set of nodes reachable in the graph E from node A in the current interpretation. Let $I = \{e(b, c), e(c, d)\}$. The set of values for X such that I is a model for the atom $\&reach[e, b](X)$ is $\{c, d\}$ since $F_{\&reach}(\{e(b, c), e(c, d)\}, b) = \{(c), (d)\}$. □

Example 5. The evaluation function of the external predicate &*rdf* is such that the atom $\&rdf[u](X, Y, Z)$ is bounded to all triples (X, Y, Z) which are in the output of $F_{\&rdf}(\Pi_1(I, u))$, for the current interpretation. The type of u is 1. E.g., if the current interpretation I is {u("`http://www.polleres.net/foaf.rdf`"), u("`http://www.gibbi.com/foaf.rdf`")}, then $F_{\&rdf}(\Pi_1(I, u))$ will return a set of triples extracted from the two specified URLs. □

Definition 2. *Let &g be a typed external predicate and $F_{\&g}$ its extensional evaluation function. Let $\bar{x} = x_1, \ldots, x_n$ be a ground input list, and let $\Pi(J, \bar{x}) = \Pi_{t_1}(J, x_1), \ldots, \Pi_{t_n}(J, x_n)$, for any interpretation J. Then: (i) &g is* monotonic, *if $F_{\&g}(\Pi(I', x)) \subseteq F_{\&g}(\Pi(I'', x))$, for any I', I'' and any $\bar{x}$, whenever $I' \subseteq I''$; (ii) &g is* anti-monotonic, *if $F_{\&g}(\Pi(I', x)) \subseteq F_{\&g}(\Pi(I'', x))$, for any I', I'' and any $\bar{x}$, whenever $I' \supseteq I''$; and (iii) &g is* linear, *if $F_{\&g}(\Pi(I' \cup I'', x)) = F_{\&g}(\Pi(I', x)) \cup F_{\&g}(\Pi(I'', x))$ for any I', I'' and any $\bar{x}$.*

Example 6. Intuitively, the &*reach* predicate is monotonic. Indeed, if we add some edge to $G' = \{e(b, c), e(c, d)\}$ so that we have, e.g., $G'' = \{e(b, c), e(c, d), e(c, h)\}$, the set of values for X such that G'' is a model of the atom $\&reach[e, b](X)$ will grow. In particular, $F_{\&reach}(G', b) = \{\langle c\rangle, \langle d\rangle\}$ and $F_{\&reach}(G'', b) = \{\langle c\rangle, \langle d\rangle, \langle h\rangle\}$. □

Many external predicates of practical interest can be classified as being monotonic. For instance, in most of the cases, dl-atoms as defined in [10] are monotonic.

Example 7. The &*rdf* predicate is linear. Indeed, let $U' = \{u$("`http://www.gibbi.com/foaf.rdf`")$\}$ and $U'' = \{u$("`http://www.polleres.net/foaf.rdf`")$\}$. Then, $F_{\&rdf}(U', u)) \cup F_{\&rdf}(U'', u)) = F_{\&rdf}(U' \cup U'', u))$, i.e., the two requested RDF sources are simply merged by union.[2] □

3.1 Dependency Information Treatment

Taking the dependency between heads and bodies into account is a common tool for devising an operational semantics for ordinary logic programs, e.g., by means of the notions of *stratification* or *local stratification* [18], or through *modular stratification* [20] or *splitting sets* [17]. In HEX-programs, dependency between heads and bodies is not the only possible source of interaction between predicates. In particular we can have:

Dependency between higher order atoms. For instance, $p(A)$ and $C(a)$ are strictly related. Intuitively, since C can unify with the constant symbol p, rules that define $C(a)$ may implicitly define the predicate p. This is not always the case: for instance, rules defining the atom $p(X)$ do not interact with rules defining $a(X)$, as well as $H(a, Y)$ does not interact with $H(b, Y)$.

[2] Note that we are assuming a simple &*rdf* predicate where entailment is not performed. HEX-programs offer the possibility to implement RDF semantics either in the language itself or by means of a different external predicate bounded to a suitable reasoner.

Dependency through external atoms. External atoms can take predicate extensions as input: as such, external atoms may depend on their input predicates. This is the only setting where predicate names play a special role.

Disjunctive dependency. Atoms appearing in the same disjunctive head have a tight interaction, since they intuitively are a means for defining a common nondeterministic search space.

Note that the above dependency relations relate *non-ground atoms* to each other rather than predicates. We next formalize the above ideas.

Definition 3. *Let P be a program and a, b atoms occurring in some rule of P. Then:*

1. *a* matches *with b, symbolically $a \approx_u b$, if there exists a partial substitution θ of variables in a such that either $a\theta = b$ or $a = b\theta$ (e.g., $H(a, Y)$ unifies with $p(a, Y)$; note that this relation is symmetric);*
2. *a* positively precedes *b, symbolically $a \preceq_p b$, if there is some rule $r \in P$ such that $a \in H(r)$ and $b \in B^+(r)$;*
3. *a* negatively precedes *b, symbolically $a \preceq_n b$, if there is some rule $r \in P$ such that $a \in H(r)$ and $b \in B^-(r)$;*
4. *a is* disjunctive dependent *on b, symbolically $a \approx_d b$, if there is some rule $r \in P$ such that $a, b \in H(r)$ (note that this relation is symmetric);*
5. *a is* externally dependent *on b, symbolically $a \preceq_e b$, if a is an external predicate of form $\&g[\bar{X}](\bar{Y})$, where $\bar{X} = X_1, \ldots, X_n$, and either*
   - *b is of form $p(\bar{Z})$, and, for some i, $X_i = p$, $t_i = \mathsf{a}$, where a is the arity of $p(\bar{Z})$ (e.g., $\&count[item](N)$ is externally dependent on $item(X)$), or*
   - *a is an external predicate of form $\&g[X_1, \ldots, X_n](\bar{Y})$, and there is some variable X_i of type a, and b is an atom of arity a (e.g., $\&DL[p, Q](N)$ is externally dependent on $q(X, Y)$ provided that Q ranges over binary predicates).*

We say that a precedes *b, if $a \preceq b$, where $\preceq = \bigcup_{i\in\{p,n,e\}} \preceq_i \cup \bigcup_{i\in\{u,d\}} \approx_i$. Furthermore, a* strictly precedes *b, symbolically $a \prec b$, if $a \preceq^+ b$ but $b \not\preceq^+ a$, where $^+$ is the transitive closure operator.*

We can now define several structural properties of HEX-programs.

Definition 4. *Let P be a* HEX*-program and $\preceq$ the relation defined above. We say that P is* (*i*) nonrecursive*, if $\preceq$ is acyclic;* (*ii*) stratified*, if there is no cycle in $\preceq$ containing some atom a and b such that $a \preceq_n b$;* (*iii*) e-stratified*, if there is no cycle in $\preceq$ containing some atom a and b such that $a \preceq_e b$; and* (*iv*) totally stratified*, if it is both stratified and e-stratified.*

For instance, the program P_{ex} from Section 2 is both stratified and e-stratified. Moreover, rules (1) and (2) form a nonrecursive program.

3.2 Dealing with Infinite Domains

Given a HEX-program P, its grounding $grnd(P)$ is infinite in general, and cannot be reduced straightforwardly to a finite portion since, given an external predicate $\&g$, the

co-domain of $F_{\&g}$ is unknown and possibly infinite. It is thus important to restrict the usage of external predicates. Such restrictions are intended to bound the number of symbols to be taken into account to a finite totality, whilst external knowledge in terms of new symbols can still be brought into a program.

Definition 5. *Given a rule r, the set of* safe variables *in r is the smallest set X of variables such that* (*i*) *X appears in a positive ordinary atom in the body of r, or* (*ii*) *X appears in the output list of an external atom* $\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m)$ *in the body of r and* $Y_1, \ldots, Y_n$ *are safe. A rule r is* safe*, if each variable appearing in a negated atom and in any input list is safe, and variables appearing in* $H(r)$ *are safe.*

For instance, the rule $r : C(X) \leftarrow url(U), \&rdf[U](X, rdf{:}subClassOf, C)$ is safe. Intuitively, this notion captures those rules for which input to external atoms can be determined by means of other atoms in the same rule. Given the extension of the predicate *url*, the number of relevant ground instances of r intuitively is finite and can be determined by repeated calls to $F_{\&rdf}$.

In some cases, safety is not enough for determining finiteness of the set of relevant symbols to be taken in account. This motivates the following stronger notion:

Definition 6. *A rule r is* strongly safe *in P iff each variable in r occurs in some ordinary atom* $b \in B^+(r)$ *and each atom* $a \in H(r)$ *strictly precedes b.*

The rule r above is not strongly safe. Indeed, if some external URL invoked by means of $\&rdf$ contains some triple of form $(X, rdf{:}subClassOf, url)$, the extension of the *url* predicate is potentially infinite. The rule

$$r' : instanceOf(C, X) \leftarrow concept(C), obj(X), url(U), \&rdf[U](X, rdf{:}subClassOf, C)$$

is strongly safe, if $concept(C)$, $obj(X)$, and $url(U)$ do not precede $instanceOf(C, X)$.

The strong safety condition is, anyway, only needed for rules which are involved in cycles of $\preceq$. In other settings, the ordinary safety restriction is enough. This leads to the following notion of a *domain-expansion safe* program. Let $grnd_U(P)$ be the ground program generated from P using only the set U of constants.

Definition 7. *A* HEX*-program P is* domain-expansion safe *iff* (*i*) *each rule* $r \in P$ *is safe, and* (*ii*) *each rule* $r \in P$ *containing some* $b \in B(r)$ *such that, for each* $a \in H(r)$ *with* $a \not\prec^+ b$*, a is strongly safe.*

The following theorem states that we can effectively reduce the grounding of domain-expansion safe programs to a finite portion.

Theorem 1. *For any domain-expansion safe* HEX*-program P, there exists a finite set* $D \subseteq C$ *such that* $grnd_D(P)$ *is equivalent to* $grnd_C(P)$ (*i.e., has the same answer sets*)*.*

Proof (*Sketch*). The proof proceeds by considering that, although the Herbrand universe of P is in principle infinite, only a finite set D of constants can be taken into account. From D, a finite ground program, $grnd_D(P)$, can be used for computing answer sets.

Provided that P is domain-expansion safe, it can be shown that $grnd_D$ has the same answer sets as $grnd_C(P)$.

A program that incrementally builds D and $grnd_D(P)$ can be sketched as follows: We update a set of *active* ordinary atoms A and a set R of ground rules (both of them initially empty) by means of a function $ins(r, A)$, which is repeatedly invoked over all rules $r \in P$ until A and R reach a fixed point. The function $ins(r, A)$ is such that, given a safe rule r and a set A of atoms, it returns the set of all ground versions of r such that each of its body atom a is either (i) such that $a \in A$ or (ii) if a is external, f_a is true. D is the final value of A, and $R = grnd_A(P)$. It can be shown that the above algorithm converges and $grnd_D(P) \subseteq grnd_C(P)$. The program $grnd_C(P)$ can be split into two modules: $N_1 = grnd_D(P)$ and $N_2 = grnd_C(P) \setminus grnd_D(P)$. It holds that each answer set S of $grnd_C(P)$ is such that $S = S_1 \cup S_2$, where $S_1 \in AS(N_1')$ and $S_2 \in AS(N_2)$. N_1' is a version of N_1 enriched with all the ground facts in $AS(N_2)$. Also, we can show that the only answer set of N_2 is the empty set. From this the proof follows. □

3.3 Splitting Theorem

The dependency structure of a program P, given by its dependency graph $\preceq$, can be employed for detecting modules inside the program itself. Intuitively, a module corresponds to a strongly-connected component[3] of $\preceq$ and can be evaluated separately. The introduction of a modular evaluation strategy would allow to use, on the one hand, different evaluation algorithms depending on the nature of the module at hand. For instance, a module without external atoms can be directly evaluated by an efficient ASP solver, whereas a specific algorithm for stratified modules with monotonic external predicates can be devised (see e.g., the evaluation strategy adopted in [7] for dl-programs). On the other hand, such a strategy would enable the evaluation of a broader class of programs, given by the arbitrary composition of modules of different nature.

A way for splitting a program in sub-modules can be given by the notion of a *splitting set* [17]. Intuitively, given a program P, a splitting set S is a set of ground atoms that induce a sub-program $grnd(P') \subset grnd(P)$ whose models $\mathcal{M} = \{M_1, \ldots, M_n\}$ can be evaluated separately. Then, an adequate *splitting theorem* shows how to plug in $\mathcal{M}$ in a modified version of $P \setminus P'$ so that the overall models can be computed.

The traditional notion of a splitting set and the associated theorem must be adapted in two respects. First, the new notions of dependency have to be accommodated. Second, we need a notion of splitting set built on non-ground programs. Indeed, given P, $grnd(P)$ is in principle infinite. Even if, under reasonable assumptions, we must take only a finite portion of $grnd(P)$ into account, this portion can be exponentially larger than P. This makes the idea of managing sub-modules at the ground level infeasible.

Definition 8. *A* global splitting set *for a* HEX*-program P is a set A of atoms appearing in P such that, whenever $a \in A$ and $a \preceq b$, for some atom b appearing in P, then b belongs to A. The* global bottom *of P with respect to A is the set of rules $gb_A(P) = \{r \in P \mid$ for each $a \in H(r)$ there is an element $b \in A$ such that $a \preceq_u b\}$.*

[3] A *strongly-connected component* (SCC) is a maximal subgraph in which every node is reachable from every other node. Note that we modify this definition and let a single node, which is not part of any SCC, be an SCC by itself.

For example, given the program P

$$triple(X, Y, Z) \leftarrow \&rdf[u](X, Y, Z), \tag{13}$$

$$C(X) \leftarrow triple(X, rdf{:}subClassOf, C), \tag{14}$$

$$r(X, Y) \leftarrow triple(X, r, C), \tag{15}$$

then, $S = \{triple(X, r, C), triple(X, Y, Z), triple(X, rdf{:}subClassOf, C), r(X, Y), \&rdf[u](X, Y, Z)\}$ is a splitting set for P. We have $gb_S(P) = \{(13), (15)\}$.

Definition 9. *For an interpretation I and a program Q, the* global residual*, $gres(Q, I)$, is a program obtained from Q as follows:*

1. *add all the atoms in I as facts;*
2. *for each "resolved" external atom $a = \&g[X_1, \ldots, X_n](Y_1, \ldots, Y_m)$ occurring in some rule of Q, replace a with a fresh ordinary atom $d_{\&g}(Y_1, \ldots, Y_m)$ (which we call* additional *atom), and add the fact $d_{\&g}(\bar{c})$ for each tuple $\bar{c} = \langle c_1, \ldots, c_n \rangle$ output by $EVAL(\&g, Q, I)$.*

For space reasons, we omit here the formal notion of a "resolved" external atom and the details of $EVAL(\&g, Q, I)$. Informally, an external atom a is resolved if its actual input list depends only on atoms in I. Thus, the input to $\&g$ is fully determined and its output can be obtained by calling $F_{\&g}$ with suitable parameters. To this end, $EVAL(\&g, Q, I)$ performs one or multiple calls to $F_{\&g}$. The external atom $\&rdf[u](X, Y, Z)$, for instance, has a ground input list. Thus $EVAL(\&g, Q, I)$ amounts to computing $F_{\&g}(\Pi_{p/1}(I, u))$. *EVAL* is more involved in case of non-ground input terms. Here, some preliminary steps are required.

Intuitively, given a program $P = \{triple(X, Y, Z) \leftarrow \&rdf[url](X, Y, Z)\}$ and the interpretation $I = \{url(\text{"}\texttt{http://www.gibbi.com/foaf.rdf}\text{"})\}$, its residual is

$$gres(P, I) = \{triple(X, Y, Z) \leftarrow d_{rdf}(X, Y, Z), \ldots,$$
$$d_{\&rdf}(\text{"}\texttt{me}\text{"}, \text{"}\texttt{http://xmlns.com/foaf/0.1/workplaceHomepage}\text{"},$$
$$\text{"}\texttt{http://www.mat.unical.it/ianni}\text{"})\}.$$

We can now formulate a generalization of the Splitting Theorem from [17].

Theorem 2 (Global Splitting Theorem). *Let P be a domain-expansion safe program and let A be a global splitting set for P. Then, $M \setminus D \in AS(P)$ iff $M \in AS(gres(P \setminus gb_A(P), I))$, where $I \in AS(gb_A(P))$, and D is the set of additional atoms in $gres(P \setminus gb_A(P), I)$ with predicate name of form $d_{\&g}$.*

Proof (*Sketch*). The idea behind the proof is that a splitting set A denotes a portion of P (viz., the bottom $gb_A(P)$) whose answer sets do not depend from the rest of the program. Also, $gb_A(P)$ is the only portion of P necessary in order to compute the extension of atoms appearing in A in any model. That is, for each model $M \in AS(P)$, we have that $M \cap G_A \in AS(gb_A(P))$, where G_A is the set of all ground instances (built from constants in C) of atoms in A. This claim can be exploited in the opposite direction as follows: We first compute $\mathcal{M}' = AS(gb_A(P))$. Then, we simplify P to P_s by removing $gb_A(P)$. The answer sets of $\{gres(P^s, M') \mid M' \in \mathcal{M}'\}$ are answer sets of P provided that additional atoms in D are stripped out. □

The above theorem is a powerful tool for evaluating a HEX-program by splitting it repeatedly in modules, which we consider next.

SPLITTING EVALUATION ALGORITHM
(Input: a HEX-program P; Output: $AS(P)$)

1. Determine the *precedes* relation $\preceq$ for P.
2. Partition the set of atoms of P into the set $Comp = \{C_1, \ldots, C_n\}$ of strongly connected components C_i of $\preceq$, and define that $C_i \prec C_j$ iff there is some $a \in C_i$ and some $b \in C_j$ such that $a \prec b$.
3. Set $T := Comp$ and $\mathcal{M} := \{\{\}\}$ (the empty model). The set $\mathcal{M}$ will eventually contain $AS(P)$ (which is empty, in case inconsistency is detected).
4. While $T \neq \emptyset$ do:
5. Pop from T some C such that for no $C' \in T$ we have $C \prec C'$.
6. Let $\mathcal{M} := \bigcup_{M \in \mathcal{M}} AS(gres(b_C(P), M))$.
7. If $\mathcal{M} = \emptyset$, then halt (inconsistency, no answer set exists).
8. $P := P \setminus b_C(P)$.

Fig. 1. Splitting algorithm

3.4 Splitting Algorithm

The class of domain-expansion safe HEX-programs encompasses a variety of practical situations. Note that such programs need not be stratified, and may harbor nondeterminism.

We can design a *splitting* evaluation algorithm for HEX-programs P under the following rationale. First of all, P is decomposed into strongly connected components. Then, a partial ordering is created between such components. Given a current set $\mathcal{M}$ of models, for each component C, we evaluate the answer sets of its possible residuals with respect to elements of $\mathcal{M}$. The actual method for computing the answer sets of each residual depends on its structure. The detailed algorithm is depicted in Figure 1.[4]

In fact, we can generalize this algorithm by popping from T a set $E \subseteq T$ of components such that for each $C_i \in E$, $\{C_j \in T \mid C_j \prec C_i\} \subseteq E$ holds (i.e., E is downwards closed under $\prec$ with respect to T). For instance, unstratified components without external atoms may be evaluated at once.

Example 8. Consider program P_{ex} from Section 2. We mimic an iteration of the splitting algorithm. The set $S = \{pick(P), drop(P)\}$ forms a strongly connected component. Assume at the moment of evaluating this component $\mathcal{M}$ contains the single answer set $M = \{proposition(p_1), proposition(p_2), axiomatic(p_2), \ldots\}$. At this stage, P no longer contains rules (1) and (2), so the bottom of S is formed by rules (3) and (4). Then, $gres(b_C(P), M)$ is the following program:

$$proposition(p_1) \leftarrow, \quad proposition(p_2) \leftarrow, \quad \ldots \tag{16}$$

$$axiomatic(p_2) \leftarrow, \tag{17}$$

$$\ldots$$

$$pick(P) \vee drop(P) \leftarrow proposition(P), \tag{18}$$

$$pick(P) \leftarrow axiomatic(P). \tag{19}$$

[4] $\mathcal{M}$ may contain exponentially many intermediate models. A variant of the algorithm avoiding this by computing one model at a time is straightforward, but omitted for simplicity.

having two answer set $M_1 = \{\ldots, pick(p_1), pick(p_2), \ldots\}$ and $M_2 = \{\ldots, drop(p_1), pick(p_2), \ldots\}$. Then, P is modified by deleting rules (3) and (4). □

3.5 Special Algorithms for Components

The above algorithm enables to exploit special evaluation algorithms depending on the specific structure of a given strongly connected component C and its corresponding residual program. In particular:

- recursive positive programs (either e-stratified or not) with monotonic external atoms can be evaluated by an adequate fixed-point algorithm as described in [7];
- programs without external atoms (either stratified or not) can be directly mapped to a corresponding ASP program and evaluated by some ASP solver (e.g., DLV); and
- generic components with generic external atoms can be evaluated by an apposite guess and check strategy (cf. [7]).

3.6 Current Prototype

The experimental prototype for evaluating HEX-programs, dlvhex, mainly follows the algorithm presented in Section 3.4. After transforming the higher-order program into a first-order syntax and decomposing it into its dependency graph, dlvhex uses an external answer-set solver to evaluate each of the components. The functions for computing the external atoms are embedded in so-called *plug-ins*, which are shipped separately and linked dynamically to the main application.

For a more detailed presentation of the implementation, we refer the reader to [9] and `http://www.kr.tuwien.ac.at/research/dlvhex`.

4 Related Work

A number of works are related to ours in different respects. We group literature into works more tailored to the Semantic Web and others of a more general perspective. The works of Heymans et al. [14, 15] on open and conceptual logic programs fall into the first group. Here, infinite domains are considered, in a way similar to classical logics and/or description logics, but adopting answer-set semantics based on grounding. The syntax of rules is restricted. In [15], call atoms are considered similar to ours, but only restricted to the propositional setting. They also consider preferences, which can be added to our framework in the future.

[10] introduces the notion of a dl-atom, through which a description-logic knowledge base can be interfaced from an answer-set program. The notion of stratification given there is subsumed by the one in this paper, given that stratified dl-programs of [10] can be viewed as an instance of HEX-programs.

Inspired by [10], Antoniou et al. [25] have used dl-atoms for a hybrid combination of defeasible and description logic, and have used in [24] an extension of dl-programs for ontology merging and alignment, which fits our framework. Early work on hybrid combination of logic programs and description logics appeared in [19, 5]. Both works do

not consider the issue of bidirectional flow of information to and from external sources and prescribe more restrictive safety conditions. Also similar in spirit is the TRIPLE language [22]. The semantics of TRIPLE considers Horn rules under the well-founded semantics. However, the information flow here is unidirectional.

Ross [21] developed a notion of stratification for HiLog programs, which basically constitute the fragment of HEX-programs without external atoms. Our notion of stratification is more general (as it handles external atoms), and no special range restrictions on predicate variables are prescribed.

The use of external atoms in logic programs under the answer-set semantics dates back to [6], where they have been modeled as generalized quantifiers. In general, this approach (based on the usual reduct by Gelfond and Lifschitz [12]) is different from ours, and no value invention has been considered there. The latter problem has been studied extensively in the database field. For instance, Cabibbo [3] studied decidable fragments of the ILOG language, which featured a special construct for creating new tuple identifiers in relational databases. He developed notions of safety similar to ours (in absence of higher order atoms) and gave conditions such that new values do not propagate in infinite chains.

In a broad sense, value invention, while keeping decidability in ASP, has been considered in [23, 2, 4]. In [23, 2], potentially infinite domains are considered by allowing function symbols, whose usage is restricted. Syrjänen [23] introduced ω-restricted programs, in which, roughly, all unstratified rules according to the dependency graph are put in some special stratum ω at the top level. Then, each function term must be bound by some predicate which belongs to some lower stratum. However, the models of ω-restricted programs have always finite positive part. They are a subclass of Bonatti's finitary programs [2], which have been designed for query answering. The work of Calimeri and Ianni [4] considered ASP with external atoms, but input and output arguments are restricted to constants (individuals), and no higher-order atoms are present. Thus, the framework there is subsumed by ours.

5 Conclusion and Future Work

We have discussed methods and techniques by which an integration of a rule-based formalism that has higher-order features and supports external evaluations, as given by the powerful framework of HEX-programs, can be made effective. In this way, declarative tool support for a wide range of reasoning applications on the Semantic Web at a high level of abstraction can be realized. For example, merging and alignment of ontologies [24], or combining and integrating different information sources on the Web in general, which possibly have different formats and semantics.

The current prototype implementation features atoms for accessing RDF and OWL ontologies, and provides a tool kit for customized external evaluation plug-ins that the user might create for his or her application. Our ongoing work concerns enhancing and further improving the current prototype, as well as extending the classes of effective HEX-programs. Finally, applications in personalized Web information systems are targeted.

Acknowledgement. This work was partially supported by the Austrian Science Funds project P17212 and the European Commission project REWERSE (IST-2003-506779).

References

1. G. Antoniou, C. V. Damásio, B. Grosof, I. Horrocks, M. Kifer, J. Maluszynski, and P. F. Patel-Schneider. Combining Rules and Ontologies. A Survey. Technical Report `IST506779/Linkoeping/I3-D3/D/PU/a1`, Linköping University, 2005.
2. P. A. Bonatti. Reasoning with Infinite Stable Models. *Artificial Intelligence*, 156(1):75–111, 2004.
3. L. Cabibbo. The Expressive Power of Stratified Logic Programs with Value Invention. *Information and Computation*, 147(1):22–56, 1998.
4. F. Calimeri and G. Ianni. External Sources of Computation for Answer Set Solvers. In *Proc. LPNMR 2005*, pp. 105–118.
5. F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. AL-log: Integrating Datalog and Description Logics. *J. Intell. Inf. Syst.*, 10(3):227–252, 1998.
6. T. Eiter, G. Gottlob, and H. Veith. Modular Logic Programming and Generalized Quantifiers. In *Proc. LPNMR'97*, pp. 290–309.
7. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. Nonmonotonic Description Logic Programs: Implementation and Experiments. In *Proc. LPAR 2004*, pp. 511–527.
8. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A Uniform Integration of Higher-order Reasoning and External Evaluations in Answer Set Programming. In *Proc. IJCAI 2005*.
9. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. dlvhex: A System for Integrating Multiple Semantics in an Answer-Set Programming Framework. In *Proc. WLP 2006*, pp. 206–210.
10. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining Answer Set Programming with Description Logics for the Semantic Web. In *Proc. KR 2004*, pp. 141–151.
11. W. Faber, N. Leone, and G. Pfeifer. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In *Proc. JELIA 2004*, pp. 200–212.
12. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
13. B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logics. In *Proc. WWW 2003*, pp. 48–57.
14. S. Heymans, D. V. Nieuwenborgh, and D. Vermeir. Nonmonotonic Ontological and Rule-based Reasoning with Extended Conceptual Logic Programs. In *Proc. ESWC 2005*.
15. S. Heymans, D. V. Nieuwenborgh, and D. Vermeir. Preferential Reasoning on a Web of Trust. In *Proc. ISWC 2005*, pp. 368–382.
16. I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, 2004. W3C Member Submission.
17. V. Lifschitz and H. Turner. Splitting a Logic Program. In *Proc. ICLP'94*, pp. 23–38.
18. T. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In *Foundations of Deductive Databases and Logic Programming.*, pp. 193–216. 1988.
19. R. Rosati. Towards Expressive KR Systems Integrating Datalog and Description Logics: Preliminary Report. In *Proceedings DL'99*, pp. 160–164.
20. K. A. Ross. Modular Stratification and Magic Sets for Datalog Programs with Negation. *J. ACM*, 41(6):1216–1266, 1994.
21. K. A. Ross. On Negation in HiLog. *Journal of Logic Programming*, 18(1):27–53, 1994.
22. M. Sintek and S. Decker. Triple - a Query, Inference, and Transformation Language for the Semantic Web. In *Proc. ISWC 2004*, pp. 364–378.
23. T. Syrjänen. Omega-restricted Logic Programs. In *Proc. LPNMR 2001*, pp. 267–279.
24. K. Wang, G. Antoniou, R. W. Topor, and A. Sattar. Merging and Aligning Ontologies in dl-Programs. In *Proc. RuleML 2005*, pp. 160–171.
25. K. Wang, D. Billington, J. Blee, and G. Antoniou. Combining Description Logic and Defeasible Logic for the Semantic Web. In *Proc. RuleML 2004*, pp. 170–181.